TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone for Armv8-M

Tomoaki Kawada¹ · Shinya Honda¹ · Yutaka Matsubara¹ · Hiroaki Takada¹

Received: 10 November 2019 / Accepted: 15 July 2020 / Published online: 27 July 2020
© The Author(s) 2020

Abstract

Control-Flow Integrity (CFI) is a class of defensive techniques against control-flow attacks such as Return-Oriented Programming. We propose a light-weight CFI scheme for RTOS-based applications, TZmCFI, which utilizes TrustZone for Armv8-M, a hardware-assisted security feature for embedded systems with tight resource constraints. TZmCFI embodies several existing CFI techniques to provide a comprehensive protection. The traditional shadow stack technique is used to ensure stack integrity and validate function returns. To protect exception handlers, TZmCFI extends *shadow exception stacks*, which are a variant of the traditional shadow stack technique we proposed in our previous work, for RTOS integration and performance improvement. We conducted an experiment on Arm Versatile Express Cortex-M Prototyping System (V2M-MPS2+) to evaluate the run-time overhead of the proposed system.

Keywords Embedded systems · Control-flow attacks · Real-time operating systems · TrustZone

1 Introduction

Arbitrary code execution is by far the most serious threat to software. It is accomplished through the following two steps: the first step, *code injection*, loads malicious code onto the target's memory, followed by the second step, *control-flow hijacking*, which transfers the control to the injected code. The common mitigation against this sort of attack is the use of a memory access policy called $W \oplus X$ (“write xor execute”), which disallows memory regions from being writable and executable at the same time, effectively thwarting code injection. $W \oplus X$ can be

✉ Tomoaki Kawada
tcpp@ertl.jp

¹ Nagoya University, Nagoya, Aichi, Japan

efficiently implemented on common hardware using a ubiquitously-available MMU or MPU.

However, there is a class of attack techniques known as Return-Oriented Programming [11]. ROP achieves a similar effect by instead of injecting new code, reusing the pieces of existing code, which are then chained together by return instructions at the end of each of the pieces with the help of a corrupted call stack. Because of this, the existing mitigation techniques for code injection are ineffective against ROP.

Control-flow integrity [1] is a class of defensive techniques against ROP attacks. However, the existing techniques mostly presume the existence of specific hardware features and even so, impose a high run-time overhead, preventing their uses in embedded systems. One of the challenges with CFI is to protect an entire software system, including an operating system kernel. The uses of RTOS in embedded systems are rather common as exemplified by the inclusion of FreeRTOS in the IoT solution provided by Amazon Web Services [2]. In a small embedded system, privileged code including an operating system kernel as well as other software components (such as a protocol stack, which requires frequent interaction with networking hardware) can comprise a substantial portion of the whole system, but existing RTOS-compatible CFI solutions including [15] relied on the trustworthiness of the privileged code and were hence unable to protect the critical CFI internal state from such components. The latest microcontrollers for small embedded systems include hardware support for a software-based trusted execution environment such as TrustZone for Armv8-M, which provides something similar to an additional privilege level that is useful for implementing security mechanisms in software. This has motivated us to develop *TZmCFI*, a light-weight CFI implementation for RTOS-based applications running on such hardware.

Unlike desktop and mobile applications which have often been the subject for previous researches on CFI, embedded applications usually include asynchronous exception handlers (including both of external interrupts and CPU-generated exceptions) as a part of them. To protect asynchronous exception handlers, we proposed *shadow exception stack* in our previous work [7]. Shadow exception stack is a variant of the traditional shadow stack technique that leverages TrustZone for Armv8-M and carefully avoids a caveat associated with Armv8-M's exception handling to protect exception handlers. In this work, we further extended this technique to support multi-tasking systems as well as to improve the runtime performance.

The main contributions of this paper are the following:

- We present *TZmCFI*, a CFI implementation for resource-constrained microcontrollers and RTOS-based applications (Sect. 3).
- We propose an improved shadow exception stack technique that supports multi-tasking systems and imposes a lower overhead (Sect. 4).
- We evaluate a prototype system based on *TZmCFI* from runtime performance and security points of view (Sect. 5).

2 Background

In this section, we present a brief background on CFI and the Armv8-M architecture.

2.1 Control Flow Integrity

In essence, CFI is a twofold system: one part being a model of valid control paths, while the other one being run-time checking code that compares the model against the actual control path. The goal of CFI is to detect invalid indirect control transfer at run-time. A CFI scheme's ability to detect such control transfers is called *precision*.

The simplest, non-trivial model is a static control-flow graph generated by static code analysis [1]. The precision of static models is limited by the conservatism of a generated control-flow graph as well as the limitation of and the difficulty in a practical pointer analysis. For example, consider a function having multiple possible callers. The caller in a single function call is just one of them, but a static model cannot express this restriction and allows the control to return to the incorrect caller.

One possible way to improve the precision is to duplicate functions for each of, or more practically, each category of function calls. [8] proposed *call graph detaching*, which enhanced the precision of static models by making a copy of functions for indirect calls, thus detaching the direct call graph from the indirect call graph. Another possible way is to incorporate a dynamic element into the model to capture the run-time behavior of the program. A *shadow stack* [1] is a data structure akin to a call stack that records a path of valid callers, created and maintained separately from the real call stack in a protected location. Other dynamic solutions include branch tracing [14], which utilizes branch recording features available in modern x86 processors, and π CFI [9], which activates edges in the CFG lazily as code pointers are generated at runtime.

Some CFI schemes only protect function calls or function returns, respectively called *forward-edge* and *backward-edge* CFI schemes. Shadow stacks only provide protection for function returns, thus they are said to be a backward-edge scheme. Being orthogonal, shadow stacks can be complemented by forward-edge ones such as [13] to encompass all indirect branches.

The dynamic check is often accomplished by patching indirect branch instructions with run-time checking code. This is usually done at build time by a modified compiler (as in the case with [13]) or as a separate build process. It is also possible to do this entirely by rewriting the compiled binary machine code through the use of a binary instrumentation technique [1, 10], although it limits access to useful high-level information such as function prototypes. Memory-layout-preserving instrumentation such as [10] alleviates various difficulties arising from memory layout modification, but imposes severe restrictions on the patched code, often requiring the use of expensive software trap instructions.

2.1.1 Protecting CFI States

CFI is a control-only protection technique, so it requires a separate mechanism to protect its own internal state data (provided that it has one). The choice of a mechanism greatly affects the overhead of dynamic CFI schemes. One of the common techniques is to use special instructions (e.g., x86 load/store with a segment selector [1]), easily distinguishable by static analysis, to access protected data. [10] leverages the hardware-assisted security mode transition provided by TrustZone for Armv8-M by utilizing the fact that function calls triggering mode transition are protected by CFI themselves.

2.1.2 System-Level CFI

Most of the existing CFI solutions including [1, 6] were designed for applications running in the user mode, assuming the operating system is trustworthy. Unlike them, we aim to protect a broader portion of a software system including the operating system and exception handlers. We call this approach *system-level CFI* to contrast with the former one, which we call *user-space CFI*. The examples of system-level CFI include [5, 10]. [15] targets embedded systems, but is not considered as system-level CFI because it depends on the trustworthiness of the operating system. System-level CFI allows the trusted computing base to be smaller. However, it requires the handling of additional events such as asynchronous exceptions and context switching.

2.2 Armv8-M

The Armv8-M architecture is the latest revision of the instruction set implemented by the Cortex-M series, Arm microprocessor cores designed for microcontrollers. One of the prominent features of Armv8-M is the addition of support for TrustZone, a technology that allows hardware-enforced isolation between trusted and untrusted software residing on a single system. Armv8-M implements TrustZone through an extension to the instruction set, called *Cortex-M Security Extensions* (CMSE).

CMSE introduces the new concept of security modes, which correspond to the Secure and Non-Secure “worlds” of TrustZone and are accordingly called *Secure* and *Non-Secure modes*. The processor determines the current security mode based on the security attribute of the memory where the currently-running code is located. Mode transitions are rigorously checked by the hardware and only permitted in a controlled manner such as through *secure gateways*, which are white-listed Secure entry points that can be called by Non-Secure code. The checks are mostly transparent to software and incur a minimal overhead, making TrustZone an excellent platform for implementing security mechanisms.

2.3 Previous Works

CFI CaRE [10] is an earlier implementation of system-level backward-edge CFI employing shadow stacks for ensuring the CFI of normal functions as well as

asynchronous exceptions. It does not address the quirk with Armv8-M's exception handling (described in Sect. 4.1) and does not support multi-tasking operating systems. SVA [4] is an LLVM-based virtual machine for running operating systems, allowing the enforcement of object-granularity memory-safety and CFI. It virtualizes the low-level architecture (e.g., loads/stores, context switching, page table modification) of the target system and the operating system accesses them through newly introduced LLVM instructions. KCoFI [5] is a SVA-based CFI implementation for FreeBSD, which achieves a lower overhead by relaxing the security policy of SVA and using lightweight instrumentation on store instructions to protect critical data structures.

3 TZmCFI

We are developing TZmCFI to investigate the applicability of a shadow-stack-based system-level CFI scheme on embedded systems.

3.1 Assumptions

We presuppose the following assumptions on the instrumented code:

Assumption 1 The code executing in Secure Mode and the generated exception trampolines are trustworthy.

Assumption 2 The target hardware does not have a vulnerability capable of defeating the isolation provided by CMSE.

Assumption 3 The instrumented code is read-only to itself.

Assumption 4 The instrumented code follows a standard ABI.

Assumption 5 The source code of the instrumented code is available.

Assumption 6 The exception vector table offset register is immutable.

Assumption 5 imposes restrictions on the choice of third-party libraries. On the other hand, it provides an additional performance advantage by allowing more-efficient ways of inserting inline checks. For example, subroutines doing runtime checks could be invoked through procedure call instructions instead of trap instructions, which usually have a significant overhead associated with exception handling. With regard to Assumption 6, Cortex-M23/33 includes an implementation option for disabling writes to the said register.

We consider the threat model in which the attacker can read and write arbitrary memory locations accessible to the code running in Non-Secure Mode. This model is commonly used in previous works on CFI. We further strengthen the model by allowing the exploitation of exception handlers.

3.2 Design

TZmCFI is a CFI implementation in which shadow stacks serve as the central part of it. TZmCFI consists of a modified compiler and a supporting run-time component, **Monitor**, which can be preloaded to the target device. Fig. 1 shows how they position themselves within the build pipeline.

The LLVM compiler infrastructure is modified to instrument indirect branches in the compiled program. For forward edges, an existing CFI mechanism built into LLVM/Clang [13] is leveraged to enforce the control flow. For backward edges, we use a multi-task-aware TrustZone-based implementation of shadow stacks that we developed for TZmCFI (described in Sect. 3.2.1).

To instrument exception handling, the exception vector table's entries are updated to point to exception trampolines (described in Sect. 4). The original exception vector table is preserved in a different location and is read by the exception trampolines to invoke the original exception handlers. Some portion of the exception trampolines resides in the Non-Secure code region, while the rest of them is implemented by Monitor.

Monitor is a software library responsible for maintaining the internal state of the CFI implementation as well as for providing means to interact with it in a controlled way. The state consists of the current task ID and shadow stacks. Protecting the state is paramount to ensure the soundness of TZmCFI, so we leverage CMSE to isolate the state from untrusted code (Fig. 2). All data accesses to the state are done through Secure functions in Monitor, all of which can be called by the instrumented code only through secure gateways. CFI guarantees that all calls to them are legal.

Monitor provides several user-callable functions that the operating system should call through manually-inserted hooks on various occasions. When creating a task, the operating system has to call a specific monitor function named `TCCreateThread` to initialize the task structure internal to Monitor with an initial program counter and obtain a task ID. However, this is vulnerable to data-oriented attacks. To protect the system against such attacks, initializing task structures is permitted only during system startup. The operating system signals that the startup process is complete through a manually inserted hook, after which code pointers generated by untrusted code are no longer trusted. This state persists until a system reset. After a system startup, the operating system must notify context switches through another hook, passing the previously obtained task ID as a parameter.

Fig. 1 The workflow for adding CFI to an application

Fig. 2 The run-time architecture of the proposed system

3.2.1 Shadow Stacks

The shadow stack instrumentation in TZmCFI is implemented as an instrumentation pass for LLVM, which borrows some implementation techniques from ShadowCallStack [12]. During the prologue/epilogue generation pass, for each function, the instrumentation pass checks if the return target is spilled to the stack. If that is the case, it inserts the following monitor calls into the function's prologue and epilogue (Fig. 3):

- *Shadow Push* inserted to a prologue, pushes a return target to the current task's shadow stack.
- *Shadow Assert* inserted to an epilogue, pops a trustworthy return target from the shadow stack, superseding the untrustworthy one from the stack. If this type of monitor call is followed by a function return instruction, they are fused into a *Shadow Assert Return* monitor call for an improved runtime performance.

The implementation of these monitor functions resides in Monitor and runs in Secure Mode, and calling them is the sole mean to interact with a shadow stack in Non-Secure Mode.

The implementation of Shadow Assert (Return) comes with two flavors: on function return, the *aborting* flavor compares the current return target against the original return target popped from the stack, and if they are different, aborts the program immediately, while the *non-aborting* flavor simply replaces the current return target with the one from the shadow stack. From a security point of view, they both provide the same CFI guarantee—a code pointer is never created from non-trusted data, though the former is likelier to detect an incorrect behavior. From

Without SS instrumentation:

```
function:
    push    {r4, r5, r6, r7, r8, lr}
    [...]
    pop     {r4, r5, r6, r7, r8, lr}
    bx     lr
```

With SS instrumentation:

```
function:
    push    {r4, r5, r6, r7, r8, lr}
    addw    ip, pc, #8 ; set continuation
    ldr.w   pc, [pc] ; jump to ...
    .word   __TCPrivateShadowPush
    [...]
    pop     {r4, r5, r6, r7, r8, lr}
    ldr.w   pc, [pc] ; jump to ...
    .word   __TCPrivateShadowAssertReturn
```

Fig. 3 The generated assembler code of the shadow stack instrumentation in TZmCFI

a performance point of view, the latter provides a lower overhead because only one copy of a return address needs to be preserved.

To support multi-tasking, each task is associated with its own shadow stack, which is stored as a part of the task structure in Monitor. When the operating system notifies the occurrence of a context switch through a Monitor hook, the current shadow stack is swapped with a new task's one.

The shadow stack instrumentation explained in this section only covers function calls and returns. For exception entries and exits, we use a different variant of shadow stack described in Sect. 4.

4 Shadow Exception Stack

We propose *shadow exception stack*, a variant of the traditional shadow stack scheme, for enforcing CFI on exception¹ returns.

Enforcing CFI on asynchronous exceptions requires the use of a shadow stack-based technique. In some sense, exceptions and function calls are very alike because they both transfer the control to some point and return to the original location using an indirect jump instruction. For exceptions, Assumptions 3 and 6 implies inline checking is required only for exception returns. However, the set of the potential jump targets of an exception return includes every executable instruction in the application, rendering static CFI schemes ineffective. For this reason, shadow stacks must be used to enforce CFI on exception returns.

To leverage a shadow stack, two pieces of code are inserted to the front and back of each exception handler. They are called *exception trampoline* and *exception return trampolines*, respectively.

There is another difference between exceptions and function calls: exception enter/exit is usually handled by a sequence hard-coded onto a processor. This poses a challenge in a sound implementation of a shadow stack, especially in the case of Arm-M as described in Sect. 4.1.

4.1 Exception Handling in Armv8-M

Armv8-M-based processors include a Nested Vectored Interrupt Controller (NVIC) for handling exceptions. Each exception source, including CPU-generated exceptions such as MPU access violation, is given an exception number. The occurrence of an event sets the corresponding exception number's pending flag, which is maintained by NVIC. NVIC compares the priority values of pending exceptions and activates the one with the highest priority value. Activating an exception raises the *current execution priority* to the exception's priority value, preventing the activation of lower-priority exceptions while allowing higher-priority exceptions to be accepted (this behavior is commonly referred to as *nested exception*) without software intervention.

¹ *Asynchronous exceptions*, or more simply *exceptions*, refer to both of software-generated traps and hardware-generated interrupts.

The current execution priority also can be manipulated through two software-accessible registers: `FAULTMASK` and `PRIMASK`. When set to a non-default value, they raise the execution priority, effectively disabling exceptions.²

Upon activating an exception, the hardware pushes the current context state onto one of stacks. The data pushed to a stack is called *exception frame* and includes the values of a subset of general-purpose registers as well as the original program counter. It updates `LR`, a general-purpose register commonly used for storing a return address, with a special value called `EXC_RETURN`. After that, the processor loads a vector address from an exception vector table and transfers the control to it. When the program performs an indirect jump to `LR` (which is exactly the same as a normal function return) and it contains `EXC_RETURN`, the processor does not simply update the program counter but instead initiates an exception return sequence, where the original context state is restored from the stack. This process utilizes information from bit fields in `EXC_RETURN`, e.g., to locate which stack the exception frame is located in.

4.1.1 Naïve Shadow Stacks

The important fact here is that the exception entry sequence only raises the current execution priority to the exception's priority value and does not entirely disable exceptions. Higher-priority exceptions can preempt the current one anytime during an exception entry sequence even before software is given a chance to execute. This undermines the soundness of a shadow stack-based CFI scheme. The point of a shadow stack is to create a copy of code pointers written on user-accessible memory before they can be corrupted by untrusted code. In this case, a higher-priority exception handler can preempt and corrupt the program counter stored in the current exception's exception frame before it is copied to a protected location by the exception trampoline (Fig. 4).

Although it is not directly supported by the hardware, it is possible to avoid the problem by disabling nested exception. This, however, increases the latency of high-priority interrupts. The proposed solution supporting nested exceptions is described in Sect. 4.2.

4.2 Proposed Solution

We propose a solution to address the issue described in Sect. 4.1. The basic idea is, whenever the exception trampoline is executed, to protect every exception frame on a stack, not just the latest one. The exception trampoline figures out and adds missing frames to the shadow stack.

Scanning a call stack to locate every active exception frame is inefficient and impractical in general. Actually, the exception trampoline only has to visit a particular subset of the stack, which can be easily found. The subset is defined in terms of a relationship, which we call *exception entry chain*. We define the relationship as two nested exception activations constitute an exception entry chain

² NMI cannot be disabled in this way.

Fig. 4 The behavior of a naïve shadow stack implementation

if they were activated successively without giving the first one's exception handler a chance to execute before the second one's exception entry sequence is started. Fig. 5 is an example where A and B form an exception entry chain. Locating an exception frame next to another is easy as long as their originating activations form an exception entry chain because, by definition, the preempted exception handler has not executed any software code, which makes the frame size predictable. For the reasons explained later in this section, the exception trampoline only has to find exception frames which are transitively related by exception entry chains to the current exception.

Exception entry chains can be considered as a parent-child relationship. Combined with a temporal relationship, this forms an ordered forest shown in Fig. 6. We call the trees in the forest *exception entry chaining trees*. It should be noted that two nested exceptions belong to separate trees if they do not form an exception entry chain (e.g., A and E in Fig. 6). The path from a node to the root of the tree it belongs to is a suffix of the exception stack, which we call *exception entry chaining stack (EECSt)*.

The rest of this section gives an explanation how the exception trampoline and exception return trampoline work in the proposed solution. Here it is assumed that software code does not corrupt exception frames. Such situations are discussed in Sect. 5.2.

For an exception activation i , the pre/postconditions of the exception trampoline (ET_i) and the exception return trampoline³ (ERT_i) are shown below:

$$\begin{aligned}
 Pre(ET_i) &: \exists n. (ShadowSt = ExceptionSt[0 : n] \wedge \\
 &\quad |ExceptionSt| - |EECSt| \leq n \leq |ExceptionSt|) \\
 Post(ET_i) &: ShadowSt = ExceptionSt \\
 Pre(ERT_i) &: ShadowSt = ExceptionSt \\
 Post(ERT_i) &: ShadowSt = ExceptionSt
 \end{aligned}$$

The exception trampoline compares the shadow stack and the exception stack and pushes missing frames to the shadow stack. $Pre(ET_i)$ guarantees that they are found as a suffix of the exception entry chaining stack Fig. 7.

³ Includes the exception return sequence in this informal proof.

Fig. 5 The behavior of the proposed shadow stack algorithm

Fig. 6 Exception entry chaining tree

Fig. 7 The behavior of the exception trampoline (compare to Fig. 6)

The exception return trampoline removes $Top(ShadowSt)$ as the upcoming exception return sequence removes the corresponding exception frame $Top(ExceptionSt)$. The integrity check between $ShadowSt$ and $ExceptionSt$ takes place here. It must check the $\min(2, |ShadowSt|)$ top elements instead of just one (elaborated in Sect. 5.2).

The proof of the preconditions $Pre(ET_i)$ and $Pre(ERT_i)$ is shown. In an exception entry chaining tree, ET_i and ERT_i are called in an alternating order:

$ET_1, ERT_1, \dots, ET_n, ERT_n$. The calling order $1, \dots, n$ obeys the post-order of the tree. Firstly, *ShadowSt* does not contain any element of *EECSt* when the exception trampoline is called for the first time in the tree. Therefore, $Pre(ET_1)$ is true. Secondly, non-EEC nested exceptions restores the stack to the original state after returning, thus $Post(ET_1) \rightarrow Pre(ERT_1)$. Finally, $Post(ERT_i)$ implies that *ShadowSt* is a prefix of *ExceptionSt* having the top element equal to the lowest common ancestor of i and $i + 1$, thus $Post(ERT_i) \rightarrow Pre(ET_{i+1})$.

We implemented the proposed shadow exception stack algorithm as a part of TZmCFI. In the current implementation, each shadow stack entry contains five fields (PC, LR, EXC_RETURN, R12, and a pointer to the exception frame). Protecting R12 is not essential as far as only exception handling is concerned. The reason R12 is included is that it is used by shadow stack instrumentation code for passing a continuation address (the return address for a monitor call, which should not be confused with the return address of the instrumented function) as a part of its special calling convention, thus corrupting it may lead to a control-flow violation.

4.3 Multi-tasking

An operating system running on Arm-M usually performs context switching by triggering a specific type of software trap called PendSV, and once being inside the exception handler, swapping the contents of the process stack pointer (PSP) register and several other registers with those of the next task which were saved when the task was suspended last time. PendSV is configured by the operating system as a lowest priority exception, so the corresponding exception frame is always stored at the location pointed to by PSP (this would not be the case if PendSV could nest another exception). Thus, swapping PSP and all of the remaining registers not included in an exception frame achieves context switching.

However, with a shadow exception stack mechanism in place, this procedure is now rejected as a security violation because the shadow exception stack mechanism does not permit the return target changing between exception entry and exit. The solution we implemented in TZmCFI is to create a shadow exception stack separately for each task and switch them when the operating system invokes the monitor hook for context switching, allowing the PendSV handler to return to the interrupted location of another task.

4.4 Performance Optimization

We implemented two kinds of optimization to alleviate the overhead of shadow exception stacks.

Accelerated Privilege Escalation When memory protection is enabled, the operating system requires a mechanism to temporarily enter Privileged mode so that kernel services invoked from a user task can access the operating system's critical data structure. One way to do this is to invoke a SVC handler, wherein the CONTROL register is updated to transition the calling task into privileged mode, but this greatly increases CPU utilization because all Non-Secure exception handlers are to be instrumented. For this reason, we took an alternative approach where we

Without TC_INTR:	With TC_INTR:
<pre> handleTimer1: addw ip, pc, #8 ; set continuation ldr.w pc, [pc] ; jump toword __TCPrivateShadowPush stmdb sp!, {r4-r9, sl, lr} [...]</pre>	<pre> handleTimer1: stmdb sp!, {r4-r9, sl} [...]</pre>
<pre> ldmia.w sp!, {r4-r9, sl, lr} ldr.w pc, [pc] ; jump toword __TCPrivateShadowAssertReturn</pre>	<pre> ldmia.w sp!, {r4-r9, sl} cpsid f ldr.w pc, [pc] ; jump toword __TCPrivateLeaveInterrupt</pre>

Fig. 8 The generated assembler code of an exception handler with and without Trampoline Shortcut

replaced this mechanism with a secure function named `TCRaisePrivilege`. For FreeRTOS, switching to this approach is as easy as to point the macro `portRAISE_PRIVILEGE` to the secure function. Assuming CFI is in place, this approach does not hinder security because CFI prevents it from being called from a disallowed location.

Trampoline Shortcut Since an exception handler is just a plain function, its return address (`EXC_RETURN` in this case) is subject to protection by shadow stack. The shadow exception stack instrumentation replaces the return address with a constant function pointer to the return trampoline. This means that it is actually unnecessary to preserve or protect the return address provided that the return instruction is replaced with a direct jump to the return trampoline. We implemented this optimization technique as a new LLVM calling convention `TC_INTR`. Fig. 8 is a concrete example that illustrates this technique.

5 Evaluation

In this section, we evaluate the proposed approach in terms of run-time overhead and security.

5.1 Overhead Analysis

We conducted an experiment to measure the run-time overhead of the proposed CFI mechanism.

The experiment was performed on Arm Versatile Express Cortex-M Prototyping System (V2M-MPS2+) [3] configured to use the FPGA image of AN505 (Example IoT Kit Subsystem Design for a V2M-MPS2+) provided by Arm. AN505 includes a Cortex-M33 processor executing at 20MHz, and an interface to the external ZBT SSRAM, where all code and data is located. Our test suite is comprised of the following three test programs: **Interrupt Latency** (Sect. 5.1.1), **FreeRTOS+MPU System Calls** (Sect. 5.1.2), and **CoreMark** (Sect. 5.1.3), all of which are bare-metal applications except for FreeRTOS+MPU System Calls.

TZmCFI Monitor and the test programs were built using the Zig programming language to facilitate the development and test process. Third-party software written in C such as FreeRTOS and CoreMark was built using LibClang 9.0 integrated into Zig's build system. Code were compiled using *ReleaseFast* and *ReleaseSmall* build

modes provided by Zig, which are hard-coded to map to the code optimization flags `-O3` and `-Os`, respectively. We modified Zig to produce LLVM bitcode output (which Clang already could do) to enable link-time optimization for all compiled code.

Throughout this section, the following abbreviations are used to refer to the various CFI mechanisms used during the experiment:

- *Ctx* The use of the context management API including task creation and context switching. This is technically not a CFI mechanism by itself but rather a prerequisite for other mechanisms. Note that even without TZmCFI, multi-tasking applications are still required to use the API (with a slimmer implementation) to correctly preempt the execution of Secure functions. Also, task deletion is never performed because it is not supported by TZmCFI.
- *SES* Shadow exception stacks (Sect. 4).
- *APE* Accelerated privilege escalation (Sect. 4.4).
- *SS* Our multi-task-aware TrustZone-based implementation of shadow stacks (Sect. 3.2.1). The two flavors of the implementation, *Aborting* and *Non-Aborting* are evaluated separately.

For the experiment, we wrote a statistical profiler that counts the number of monitor calls that take place while running benchmarking code. The profiler accounts the following types of monitor calls:

- *EntInt* and *LeaInt* represent the execution of an exception trampoline and an exception return trampoline. This pair of operations is executed every time an exception is handled to keep track of valid exception return targets on SES.
- *ShPush* and *ShAsrt* represent push and pop operations on SS, which are inserted to function prologues and epilogues by the compiler. Some leaf functions do not have them if the return target is not spilled to memory.
- *ShAsrtRet* is a fused operation of *ShAsrt* and a function return, used for performance optimization.

5.1.1 Interrupt Latency

We created a benchmark program to assess the impact on the interrupt response time. Two timers, *Timer0* and *Timer1*, both having interrupt handlers, were configured to fire almost simultaneously. The interrupt priority of *Timer1* was set to higher than that of *Timer0*. We measured the interrupt response time of *Timer1* by reading the timer value in *Timer1*'s interrupt handler.

The timing difference between them was controlled by the *skew* parameter. The variation in the skew changes the behavior of the handling order of the interrupts and the code path taken by the exception trampoline's implementation. For example, a positive skew causes *Timer1* to fire while *Timer0*'s handle is still running, in which case the exception trampoline iterates through the stack until it encounters *Timer0*'s exception frame. On the other hand, a negative one makes *Timer1* the top-level exception, in which case the exception trampoline proceeds to the code path where it just pushes every frame in the exception entry chain stack. A

Fig. 9 The interrupt response time of Timer1

Table 1 The interrupt response time of Timer1

Skew	ReleaseFast		ReleaseSmall	
	Uninstrumented	Instrumented	Uninstrumented	Instrumented
< -4	35	132 (+ 97)	31	131 (+ 100)
2	30	126 (+ 96)	25	125 (+ 100)
7	36	236 (+ 200)	31	233 (+ 202)

All numbers are in cycles

specific skew value induces an exception entry chain, in which case the exception trampoline saves the exception frames of both of Timer0 and Timer1. We swept the skew parameter through a range and observed the changes in the interrupt response time.

Figure 9 and Table 1 show the result of the measurement. The overall increase in the interrupt response time in the cases where the exception trampoline ran only once ($skew < 6$) was 97 cycles. In the cases where Timer1 fired late ($skew \geq 6$), the increase was twofold due to the exception trampoline being Timer0 handler's critical section.

In this experiment, we could not find a datapoint indicating the presence of an exception entry chain. To obtain the interrupt response time of the case including the occurrence of an exception entry chain, we performed another experiment identical to the previous one except that the `cpsid f` instruction that disables interrupts was swapped with the next instruction. An exception entry chain was observed at $skew = 7$ and the interrupt response time was 140 cycles (*ReleaseFast*) and 142 cycles (*ReleaseSmall*). Rewriting the program to use the FPGA RAM block inside the FPGA yielded little improvement. Because the algorithm's most time-consuming part is expected to be the stack walk loop, this result suggested that the overhead was largely caused by inefficiencies in the invocation of the core subroutines, not by the algorithm.

Table 2 The execution time for each type of FreeRTOS system calls

Build mode	Ctx	SES	APE	Shadow stack	Overhead ^a	NewTask	DelTask	NewTask+disp	DelTask+disp	SemTake	SemGive	SemGive+disp
ReleaseFast	✓				18	1026	347	2803	429	175	210	561
	✓				18	3050	347	4927	529	175	210	661
	✓		✓		18	3012	309	4888	492	137	171	622
	✓	✓			18	3228	526	5284	887	354	389	1019
	✓	✓	✓		18	3011	309	5066	671	137	171	801
	✓	✓		Non-aborting	59	3517	747	5575	1051	482	518	1229
	✓	✓	✓	Non-aborting	59	3258	488	5315	792	222	258	969
	✓	✓		Aborting	62	3538	762	5593	1063	491	527	1244
	✓	✓		Aborting	62	3276	500	5330	801	228	264	981
	✓				19	1279	359	4003	429	177	210	561
ReleaseSmall	✓				19	4065	359	6889	529	177	210	661
	✓		✓		19	4027	321	6851	491	139	171	622
	✓	✓			19	4242	537	7244	885	355	388	1017
	✓	✓	✓		19	4026	321	7028	669	139	171	800
	✓	✓		Non-aborting	59	4616	844	7606	1072	481	517	1230
	✓	✓	✓	Non-aborting	59	4357	586	7347	813	222	258	971
	✓	✓		Aborting	62	4643	865	7627	1087	490	526	1245
	✓	✓	✓	Aborting	62	4381	604	7365	825	228	264	983

All values within are shown in cycles

Fig. 10 The relative overhead for each type of FreeRTOS system call. The build mode used here is *ReleaseFast*

5.1.2 FreeRTOS+MPU System Calls

We constructed a benchmark application to measure the execution times of FreeRTOS’s API functions. All functions are called from restricted tasks (i.e., tasks with memory protection) executing in Unprivileged mode.

Table 2 shows the measured execution times of FreeRTOS’s API functions with various CFI mechanisms turned on or off. The interesting part is shown as a bar chart in Fig. 10. Table 3 shows the number of monitor calls taking place during FreeRTOS system calls.

Task creation and deletion are intended to be done only during system initialization, and task deletion is not supported and treated as no-op (hence the lack of contribution of Ctx to DelTask). Even uninstrumented, the “Ctx” portion in an orthodox application is not exactly zero (but usually less than TZmCFI’s Ctx) for the reason explained earlier in Ctx’s definition. Since the execution times of task creation/deletion functions are dominated by Ctx, this fact makes it challenging to evaluate the overhead in a fair manner. Nevertheless, their results are still shown here for completeness.

SES added about 180 cycles for each triggered exception. All measured API functions trigger a software trap for entering Privileged mode. APE is shown to be an effective approach to eliminate the extra overhead caused by this. Non-aborting shadow stacks were marginally faster than aborting shadow stacks. The final overall overhead with all CFI mechanisms and optimization techniques turned on was 22–84%, varying primarily based on whether a context switch occurred or not.

Table 3 The number of monitor calls for each type of FreeRTOS system calls

Build mode	Event	Overhead ^a	NewTask	DelTask	NewTask+disp	DelTask+disp	SemTake	SemGive	SemGive+disp
ReleaseFast	EntInt	0	1	1	2	2	1	1	2
	LeaInt	0	1	1	2	2	1	1	2
	ShPush	1	8	6	9	5	4	4	6
	ShAsrt	0	1	3	1	1	1	1	1
	ShAsrtRet	1	7	3	6	4	3	3	5
ReleaseSmall	EntInt	0	1	1	2	2	1	1	2
	LeaInt	0	1	1	2	2	1	1	2
	ShPush	1	10	8	11	5	4	4	6
	ShAsrt	0	1	4	1	1	1	1	1
	ShAsrtRet	1	9	4	7	5	3	3	5

^aThe raw measurement for an empty code section. This value is already subtracted from the rest of the measurements

Fig. 11 The CoreMark scores

Table 4 The per-iteration event statistics for CoreMark

Build mode	Event	Count	Build mode	Event	Count
ReleaseFast	ShPush	1249	ReleaseSmall	ShPush	1671
	ShArtRet	1249		ShArtRet	1671

5.1.3 CoreMark

Fig. 11 shows the CoreMark scores (iterations per second). Table 4 shows the number of monitor calls taking place during a single iteration of CoreMark. The measured overhead for the ReleaseFast build was 12%. ReleaseSmall fared worse, most likely because of more conservative function inlining, which increased the number of shadow stack operations.

Based on this result, we can estimate the number of cycles spent for every pair of shadow stack operations. For ReleaseFast + Non-Aborting SS, it is calculated as:

$$\frac{SystemCoreClock}{ShPushCountPerIteration} \cdot \left(\frac{1}{Score} - \frac{1}{BaselineScore} \right) \approx 41.73 [\text{cycles}]$$

The result is somewhat more favorable than [15], which reported a decrease of about 21% in the CoreMark score. The improvement is attributed to the use of a CMSE secure gateway in place of an expensive software trap to enter the processor mode to access a shadow stack. [10] provides results for Dhrystone and two custom microbenchmarks but not for CoreMark. Their result indicated a 513% increase in Dhrystone's execution time, even though the monitor was called only for 34 times during a single run, which took 0.15 milliseconds uninstrumented. These numbers amount to about 900 cycles spent for every pair of shadow stack operations, which is 20 times larger than the estimation for our solution calculated by the formula above. The overhead of a protected shadow call stack reported by [1] varies across different programs in range 5–21%, which is close to (or lower than, if the result from Sect. 5.1.2 is also taken into account) our result.

5.2 Security Analysis

A memory region is considered to be in the *tainted* state if potentially-malicious code with a write permission on that memory region was executed. A tainted memory region can be reverted (decontaminated) to the clean state by checking the contents against a secure copy created when it was known to be clean. Based on our threat model, this means that the entirety of the memory space writable by Non-Secure code is tainted every time untrusted code is executed.

A shadow exception stack-based CFI scheme must maintain the following invariants:

- Invariant 1 A tainted exception frame is not inserted to a shadow stack.
- Invariant 2 The processor's exception return sequence does not read a tainted exception frame.
- Invariant 3 The control flow of the exception trampoline and the exception return trampoline is not based on tainted data.

The exception return trampoline compares $Top(ShadowSt)$ against the actual top exception frame and should there be a discrepancy, it registers it as a security violation and terminates the program. Therefore, Invariant 1 implies Invariant 2. The exception trampoline scans the stack for new exception frames, which are not tainted by definition, hence Invariant 1. However, the algorithm looks ahead by *one frame* to determine if the stack traversal should be terminated. If the exception return trampoline were to check only the top frame, thus leaving an unchecked frame on the top, it would break Invariant 3. The proposed algorithm avoids this caveat by checking the $\min(2, |ShadowSt|)$ top frames.

6 Conclusion

With the emergence of IoT and smart devices and the increase in the complexity of embedded systems, it is increasingly important to employ defensive security techniques such as CFI on embedded applications as well. However, the state of its adoption on embedded systems is still limited. In this paper, we introduce *shadow exception stacks*, a CFI mechanism for asynchronous exceptions with support for nested exceptions, for which a naïve shadow stack implementation only provides incomplete protection. We integrated the shadow exception stack mechanism into our prototype system, and conducted a performance evaluation. The measured overhead is moderate if not significant, although in some cases the overhead can be avoided by replacing software traps with CMSE gateway function calls. Finally, we present an informal proof that the proposed algorithm can ensure the control-flow integrity of exception returns, thus our system-level CFI solution provides a comprehensive protection.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this

article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Abadi, M., Budiu, M., Erlingsson, U.: Control-flow integrity. In: ACM Conference on Computer and Communication Security (CCS), Alexandria, VA, pp. 340–353 (2005) <https://www.microsoft.com/en-us/research/publication/control-flow-integrity/>
2. Amazon Web Services, Inc (2020) FreeRTOS—real-time operating system for microcontrollers—AWS. <https://aws.amazon.com/freertos/>. Accessed 30 Mar 2020
3. Arm Ltd (2019) Arm MPS2+ FPGA prototyping board. <https://www.arm.com/products/development-tools/development-boards/mps2-plus>. Accessed 14 May 2019
4. Criswell, J., Lenharth, A., Dhurjati, D., Adve, V.: Secure virtual architecture: a safe execution environment for commodity operating systems. In: SOSP '07: Proceedings of the Twenty First ACM Symposium on Operating Systems Principles (2007)
5. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: Complete control-flow integrity for commodity operating system kernels. In: 2014 IEEE Symposium on Security and Privacy (SP), IEEE, pp. 292–307 (2014)
6. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., Sadeghi, A.R.: MoCFI: a framework to mitigate control-flow attacks on smartphones. In: Network and Distributed System Security Symposium (NDSS) (2012)
7. Kawada, T., Honda, S., Matsubara, Y., Takada, H.: Shadow exception stacks: control-flow integrity for asynchronous exceptions using TrustZone for Armv8-M. In: Proceedings of the 6th International Embedded Systems Symposium (IESS 2019) (2019)
8. Moreira, J.B.C.G.: Protection mechanisms against control-flow hijacking attacks. PhD thesis, University of Campinas (2016)
9. Niu, B., Tan, G.: Per-input control-flow integrity. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA, CCS '15, pp. 914–926 (2015) <https://doi.org/10.1145/2810103.2813644>
10. Nyman, T., Ekberg, J.E., Davi, L., Asokan, N.: CFI CaRE: hardware-supported call and return enforcement for commercial microcontrollers. In: Dacier, M., Bailey, M., Polychronakis, M., Antonakakis, M. (eds.) Research in Attacks, Intrusions, and Defenses, pp. 259–284. Springer, Cham (2017)
11. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM conference on Computer and communications security, ACM, pp. 552–561 (2007)
12. The Clang Team (2020) ShadowCallStack—Clang 11 documentation. <https://clang.llvm.org/docs/ShadowCallStack.html>. Accessed 30 Mar 2020
13. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity in GCC & LLVM. In: 23rd USENIX Security Symposium (USENIX Security 14), USENIX Association, San Diego, CA, pp. 941–955 (2014) <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice>
14. van der Veen, V., Andriess, D., Goktas, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H., Giuffrida, C.: Practical context-sensitive CFI. In: CCS 2015—Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery (ACM), vol 2015-October, pp. 927–940, <https://doi.org/10.1145/2810103.2813673>
15. Walls, R.J., Brown, N.F., Le Baron, T., Shue, C.A., Okhravi, H., Ward, B.C.: Control-flow integrity for real-time embedded systems. In: 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.